APPENDIX A

On Chip Multiprocessor指向 制御並列アーキテクチャ MUSCATの提案

鳥居 淳†　近藤 真己§　本村 真人‡　西 直樹†　小長谷 明彦†

NEC †C&C研究所 ‡マイクロエレクトロニクス研究所
§NEC情報システムズ

概要

1チップ上に複数のPE (Processing Element) を集積することを前提にした制御並列処理アーキテクチャ MUSCAT (MUlti-Stream Control ArchiTecture) を提案する。MUSCATでは、従来並列処理が難しいと言われていた逐次依存性が多い問題についても、制御スペキュレーション／データスペキュレーションの2種類のスレッド投機実行を行うことによって性能向上を可能にした。これらは拡張命令によって実現され、従来プロセッサとの命令上位互換性が確保される。シミュレーションによる評価の結果、1PEモデルと同一キャッシュメモリを持った8PEモデルにおいて、逆離散コサイン変換で逐次実行の約3.0倍、ファイル圧縮プログラムで約1.7倍の性能向上が確認された。

Control Parallel On-Chip Multi-processor: MUSCAT.

Sunao Torii†　Masaki Kondo§　Masato Motomura‡　Naoki Nishi†　Akihiko Konagaya†

NEC †C&C Res. Labs. ‡Microelectronics Res. Labs.
§NEC Informatec Systems, Ltd.

Abstruct

We have developed a control parallel mutlithreaded architecture: MUSCAT. This report presents its basic idea, instruction set architecture and elementary evaluation. MUSCAT has several architecture mechanisms to support the parallel execution, namely, fast thread creation, inter-thread data-inheritance, and speculative thread execution. These mechanisms are driven by special instructions that are newly introduced onto an otherwise conventional instruction set. Our preliminary evaluation results are obtained by using a trace-driven pipeline simulator have shown that a eight processing-element microprocessor achieves 3.0 speedup for an IDCT(inverse fast discrete cosine transform) program, and 1.7 speedup for a Compress program.

1　はじめに

ここ数年のマイクロプロセッサの性能向上は、周波数の向上と共に高度な命令レベル並列処理の実現によってなされてきた。しかしながら、命令レベル並列処理は、理論的な性能の限界に近づき、徐々にリソース増加に対して充分な効果が得られなくなりつつある。一方、VLSIの集積度の飛躍的な向上により、1チップに複数のPE(Processing Element)を集積するオンチップマルチプロセッサが実現可能になりつつある。

このようなことから、我々はオンチップマルチプロセッサ向けのアーキテクチャVIRC(Virtual Register Set Computer)を提案し、評価してきた[1][2]。VIRCでは順序付きマルチスレッド実行モデルを提案し、スケジューリング、メモリ管理オーバヘッドが従来マルチスレッドモデルに比べ大幅に低減できることを確認した。しかしながら、このような並列環境では、従来の逐次プログラムに対してソースレベルの並列化の必要性や、依存性の強いコードに対しては有効な並列化が難しいなどの問題があり、数値計算などの特定用途を除けば従来のプロセッサからの移行は困難であると言わざるを得なかった。

我々はこれらの背景から、オンチップマルチプロセッサは以下の要求を満たすことが必要と考える。

並列処理の汎用性
　性能向上を可能とする適用範囲を広げる。

コーディングの容易性
　逐次コードの自動並列化コンパイラを実現する。

将来環境への適応性
　並列プログラミング環境に対応する。

Joint Symposium on Parallel Processing for 1997, Society of Information Processing, Collections of Imformation Processing Symposium Lectures, May 28, 1997, pp. 229-236

On Chip Multiprocessor Control Parallel Architecture MUSCAT Proposed.

By Sunao Torii[†], Masaki Kondo [§], Masato Motomura [‡], Naoki Nishi [†] and Akihiko Konagaya [†]

NEC [†] C&C Research Laboratory, [‡] Microelectronics Research Laboratory, and [§] NEC Informatec Systems, Limited

(Refer to the original text for an English abtsract on p. 229.)

1. Introduction

During the past several years, the improvement of the efficiency of the microprocessors has been effected by an improvement in frequency as well as by the realization of instructional level paralell processing. Nevertheless, the instructional level parallel processing has come close to the theoretical limits of its efficiency, with a consequence that the acquisition of satisfactory effects in view of the increasing resources is becoming gradually difficult.

In the meanwhile, the on-chip multiprocessor which accumulates a plurality of processing elements (PE) on one single chip has come close to realization.

With the above in the background, we have proposed a VIRC (Virtual Register Set Computer) or an architecture for the on-chip multiprocessor and offered its evaluations (1), (2). It has been confirmed that the VIRC has proposed an orderly multi-thread execution model, thereby making it possible to drastically reduce the scheduling and memory control overhead as compared with the conventional nulti-thread models. In such a parallel environment, however, there are such problems as the necessity for the parallel source levels for the conventional successive programs and the difficulty of an effective parallel arrangement for those codes which have a high level of dependency. As a result, it has been decided that any shift from the conventional processor is difficult except for such specific usages as numerical calculations, etc.

With the above in the background, it is our belief that the on-chip multiprocessor will have to meet the following requirements:

Wide Use of Parallel Processing: The applicable range that enables efficiency improvement will have to be widened.

Easy Coding: Automatic parallel compiler for successive coding will have to be realized.

Adaptability to Future Environments: A parallel programming environment will be coped with.

We propose a MUlti-Stream Control Architecture MUSCAT which meets the above-mentioned requirements. In this MUSCAT, the method of control parallel processing that utilizes the advantage of the on-chip multiprocessor with its short distance between PE's and its low cost of communications is introduced.

In the following report, the authors will concentrate their explanation on the fundamental architecture of MUSCAT and the result of its evaluation. In Chapter 2, the fundamental architecture of MUSCAT as well as the control parallel processing will be described, followed by Chapter 3, where the proposed instruction set will be explained. The method of producing a code that conforms to this instructional set architecture will be explained in Chapter 4, with its evaluation being made in Chapter 5. The information that has thus been obtained will be summarized in Chapter 6. This will be followed by Chapter 7, where related researches will be described and the problem facing us in the future will be described in Chapter 8.

2. MUSCAT Fundamental Architecture

2.1. Control Parallel Processing

In order to expand the applicable range of the multi-thread parallel processing, it is essential to be able to carry out the parallel processing with finer grain size. Therefore, parallel processing based on the control parallel processing is introduced into MUSCAT. According to this control parallel processing, the control flow of the program is analyzed, a pass which will be executed in the future is anticipated and parallel processing is carried out ahead of it.

In the case of the control flow of a fundamental block which is shown in Figure 1, for example, the execution of the fundamental block E is confirmed at the time when the execution of the fundamental block A is confirmed and it is executed. The unit thread for paralelling becomes a macro basic block unit that has summarized the basic block or a plurality of basic blocks. It goes without saying that, for parallel execution, it becomes necessary to erase the data-dependency amongst these threads.

2.2. Thread Model

In the multi-thread parallel processing of fine-grain threads, it is desirable to effect the execution by eliminating the paralleling overhead as much as possible as compared with the conventional cases. Accordingly, the following hardware support is carried out in the case of MUSCAT:

1. Incorporation of Thread Control into the Instruction Set.

(Figure 1: Fundamental Block Control Flow Graph on p. 230.)

(Figure 2: Control Parallel Processing and Fork One-Time Model.on p. 230. a. Order of Successive Executions. b. Successive Execution. c. Control Parallel Processing.

d. Thread production. e. Dependency. f. Dependency. g. Fork One-Time Model. h. Thread production. i. Thread production. j. Dependency. k. Dependency.)

2. Thread Control Hardware

3. Succession of the Register Contents at the Fork Time.

For the purpose of reducing the thread management overhead, a thread control instruction is directly added to the instruction set and, at the same time, the fork one-time model which is capable of producing a child thread once, at most, is introduced by the thread during its life-time. In view of the fact that the control parallel processing is parallel processing wherein the successive order relationship is firmly maintained, it is believed that there is little effect of the parallelism by this model. (Reference is to be made to Figure 2.)

In addition, the dependency of control or data is limited to one single direction from the parent thread to a child thread. Because of this, it becomes possible to effect successive execution according to which the execution of a child thread is initiated after the completion of the parent thread.

On the other hand, it is not possible for the parent thread to receive the data that the child thread has produced and to wait for the completion of the child thread. The normal dependency relationship of data is produced in the case where the parent thread becomes a producer of the data and the child thread a consumer of the data.

Because of the strong restrictions of the model as described above, thread management becomes drastically simplified. Specifically, this simplification includes the following advantages:

. The number of simultaneously existing threads is less than (PE number plus 1).

. Generation definition can be made for each thread.

. Deadlock free is guaranteed.

. The fork tip is limited to the neighboring PE.

. No special simultaneous mechanism is required.

Thus, it becomes possible to make hardware for thread management. Nevertheless, there is a possibility for problems to arise, depending upon the nature of the problems involved, such as the restrictions of the parallelism and the dispersion of the load, etc.

Next, a register value succession mechanism at the time of thread production is introduced so as to avoid the memory access or message transmission and receiving overhead that will increase due to the parallel processing, thereby reducing the data transmission cost amongst the threads. This is for the purpose of causing the child thread to copy the registered value of the parent thread at the time when the thread is produced. After the production of the child thread, the change in the registered values of the parent thread and the child thread becomes independent, with no hand-over of the data between the threads by using the register being carried out.

For the parallel execution of the prior threads, moreover, it is necessary for the data not to be dependent among the threads even in the case where, as far as control is concerned, the execution has been confirmed. In the MUSCAT, handling is devided between the case where either the presence or absence of said dependency is statically solved and the case where it is not, on the premise of dependency analysis by means of a compiler. The details will be shown in Section 3.2.

2.3. Thread Speculation

The basis of the control parallel processing is that the preceding threads on which execution has been confirmed are executed in parallel. In the actual practice, however, there are many cases where no sufficient amount of threads on which execution has been confirmed is obtained.

In addition, there is a possibility for the rate of parallelism to become low, thereby making it difficult to achieve the desired efficiency, due to such causes as the dependency which is dynamically determined or the limits of the compiler analysis. Because of this situation, the following two speculations are introduced into MUSCAT, thereby supporting the speculative execution of the threads in terms of the hardware.

Control Speculation:

A thread whose execution possibility is high is speculatively executed prior to the confirmation of the execution.

Data Speculation:

Load/Store is speculatively executed on the assumption that there is no data dependency among the threads.

Regarding the threads which are in a speculative state, a tentative execution is carried out in the range where the cancellation of execution is possible in terms of hardware. In the case where the control speculation has failed, the execution of the child thread and thereafter will be abandoned.

In the case where it has become apparent that there has been a collision in the address to be accessed, the data speculation is considered a failure, with a result that, on the return of the registered value, a re-execution of the instructions subsequent to the load instruction where the collision took place is carried out.

3. Instruction Set Architecture

In MUSCAT, the control over the parallel action of the thread as described earlier is realized by expansion against the ordinary instruction set. Table 1 shows the expansion instructions of MUSCAT.

--

Table 1: List of MUSCAT Expansion Instruction Sets.

Instruction	Meaning
Thread Production Instructions	
FORK	Control confirmation fork
SPFORK	Control speculative fork, Fork tip Static designation
Thread Ontrol Instructions	
THFIX	Child thread execution confirmation
THABORT	Child thread execution termination
TERM	Self-thread completion
TERM##	Conditions Established: Child thread execution confirmation, self thread completion Condition Non-Established: Child thread abandonment, Self thread execution continuation
Data Dependency Control Instruction	
BLOCK	Child thread designated address access prohibition
RELEASE	Designated address access ban lifting
DSPIN	Child thread data speculation designation
DSPOUT	Child thread data speculation lifting

Simultaneous Instructions

CWAIT	Wait for Self-thread execution confirmation state
PWAIT	Wait for parent thread completion

--

## indicates that word showing the conditions such as Equal, Greater Than, etc. is inserted.

3.1. Thread Control Instruction

The thread is produced by a thread production (fork) instruction. In the case of the fork instructions, two kinds including the FORK instruction for confirmed forks and the SPFORK for speculative forks are prepared. When a child thread has been produced by means of the SPFORK, either the child thread is confirmed by means of THFIX or the thread is abandoned by THABORT on the side of the parent. At the time when the condition has been established, TERM ## confirms the child thread and completes the parent thread. After the production of a thread, the parent thread is terminated by means of a TERM instruction. (Reference should be made to Figure 3.)

In the case where the parent thread that has been SPFORKed is TERMed, the execution of the child thread is confirmed at that point.

Even though the thread production is limited to one at most by means of the fork one-time model, the fork becomes possible once again when the control speculation form is carried out and the speculation has been failed.

(Figure 3: Control Speculation on p. 231. A. Successful Control Speculation. B. Failed Conrol Speculation. a. Unconfirmed state. b. Unconfirmed state.)

3.2. Data Dependency Control Instruction

The data dependency relationship on the memory can be cancelled by designating some suitable control method by employing a data depenendecy control instruction prior to the child thread fork. This can be realized by carrying out the division on the basis of the analytical result of the static data dependency relation and selecting a control instruction in the following cases:

In the Case Where Dependency is Obvious:

A BLOCK instruction that prohibits access to a specific address is inserted.

In the case of a possibility of dependency:

A DSPIN instruction is inserted, instructing that the child thread act by means of data speculation.

In the Case Where Non-Dependency Can be Guaranteed:

No control instruction whatsoever is added.

BLOCK instruction for prohibiting access to a specific address or the DSPIN instruction for data speculation designation are both set/released on the side of the parent thread, with the effectiveness being relayed from the child thread to the grandchild thread and thereafter. Accordingly, there is no need to analyze the memory operations of those generations below the child thread.

Data speculation is used in the case where the address of the store that exists after the fork of the parent thread has not been determined at the time of the fork. In this case, the DSPIN instruction is executed prior to the fork and the data speculation is released after the completion of the store in the DSPROUT.

The load of the grandchild thread will not be confirmed unless both the parent and the child DSPROUT. (Refer to Figure 4.)

(Figure 4: Data Speculation on p. 232. a. State of data speculation. b. Confirmed instruction upon the DSPROUTing of the thread (T0). c. Instruction confirmation upon the DSPROUTing of both the thread (T0) and the thread (T1).)

In the case of the MUSCAT, moreover, the reverse dependency relation where, prior to the memory load of the parent thread, a store operation is carried out for the same address on the side of the child thread is dealt with as follows:

The write-in of the child thread is all stored in the store buffer and the re-writing of the main memory is delayed until it becomes a parent thread by means of hardware, thereby guaranteeing the order relation.

These functions become necessary for the sake of control parallelism; however, they serve as a hindrance in the case where the parallel code for a multiprocessor of the symmetrical type is caused to act. Therefore, there is prepared a mode whereby no action takes place by the page unit of the TLB entry, with the normal parallel code action being thus supported.

3.3. Synchronization Instruction

Since the parallel processing is carried out in the range where the successive order relationship is maintained in the case of MUSCAT, the synchroneity between the threads is tacitly carried out. In view of the fact that there are cases where it is more advantageous to effect synchroneity because of such relationship as the dependency of the data, etc., a clear-cut synchroneity instruction is prepared in readiness.

CWAIT is an instruction to wait until the execution of the self thread becomes certain. It prevents the action until it can be guaranteed that there will be no cancellation of such an access that will create a problem if cancelled in such cases as the write-in for specific resources, etc.

In addition, PWAIT can wait until the parent thread is completed and the self thread becomes a parent thread. When it is employed in the case where it is obvious that the self thread can access a resource that the parent thread can access, it becomes possible to eliminate the data dependency control instructions.

4. Method for Code Preparation

4.1. Fork Policy

When a thread abort takes place in a control speculation, the effect is great indeed as it is propagated not only to the child thread but also to the generations after the grandchild thread. To improve the efficiency, it is necessary to reduce such conditional branchings as are shown in Figure 5(a) as much as possible, with an eye toward a reduction of speculations as much as possible.

As it is possible in such a case to execute the basic block E irrespective of the branching of the fundamental blocks A and C, the basic block E can continue the execution even if the branching of the basic blocks A and C may fail. In order that the child thread may succeed in the values produced by the fundamental blocks A, B, C and D, there is a need to have a memory, thereby necessitating either a data dependency control instruction or a synchronous instruction. Accordingly, there are cases where it is more advantageous to prepare the code so as to carry out the forking by using the register succession after the confirmation of the value that is to be succeeded at the time when the granular degreee of the thread happens to be small. This serves as a trade-off in code production.

(Figure 5: Policy for the Determination of the Fork Locations.)

Figure 5(b) shows a case involving a loop thread development. The loop has a sufficient amount of parallelism and this indicates processing which seems to show the largest improvement of efficiency.

As is shown in Figure 6, forking is carried out after the confirmation of the register value that is to be relayed between the loops as is shown in Figure 6. In the case where the number of the repetitions in a static state is unclear, the loop is constituted as a thread in a state of control speculation by using the SPFORK.

(Figure 6: Threading of the Loop on p. 233. A. Number of the repetitions can be statically analyzed. B. Number of the repetitions cannot be statically analyzed. a. (Translator's Note: One character is unclear) register confirmed. b. (Translator's Note: One character is unclear) register confirmed.)

4.2. Handling of Data Dependency

After the determination of a location candidate for the insertion of a fork instruction, the control instruction for data dependency on the memory is inserted. This is to investigate the dependency of the addresses in the memory store operations on all control passes between the location of the fork to the TERM instruction and the THABORT instruction, with a data dependency control instruction inserted to the side of the parent thread or a synchroneity instruction to the child thread side.

4.3. Guarantee of the Fork One-Time Model

After the determination of the fork location, measures are taken to guarantee the fork one-time model. As is shown in Figure 7, this is effected by analyzing all of the control flows that are possible after the fork instruction of the parent thread and inserting a thread completion instruction at the final spot where the successive code flow is connected to the child thread and inserting a thread abort instruction to the spot where the lack of connection to the child thread has been clarified.

(Figure 7: Guarantee of Fork One-Time Guarantee. A. In the case of a fork instruction. B. In the case of a SPFORK instruction.)

5. Evaluation

5.1. Evaluation Model

For the purpose of an evaluation of the efficiency of the MUSCAT, a hardware simulation model shown in Figure 8 was prepared.

(Figure 8: MUSCAT Simulation Model (4PE Construction) on p. 233. a. Thread management unit. b. Instruction cache. c. Instruction decoder. d. Instruction window. e. Register file. f. (Translator's Note: Illegible ... except for A ... Reservation station). g. Road store reservation station. h. (Translator's Note: Illegible except for buffer.) i. (Translator's Note: Illegible except for address (dependency?) analysis.) j. Data cache. k. System interface. m. System pass.)

The operating unit and the road store units are so arranged to be on a co-ownership basis in view of the fact that they are on chip. In view of the fact that the control parallel processing can utilize the parallelism of the instruction level parallels, each PE has been enabled to decode or complete four instructions/one cycle in conjunction with the latest super scalar processor, thereby supporting the branching speculation in the PE.

In addition, the production of the thread has been carried out for the adjacent PE in such a way as can be executed in one cycle. The threads which were in a state of data speculation, like the branch speculation, made it possible to tentatively execute the number of instructions equivalent to the instruction window size from the first memory operation.

A pipe line simulation on the trace level which was in conformity with this simulation model was prepared. This simulator carries out the simulation of the control parallel processing by collecting the traces of the successive codes beforehand on a work station and obtaining a coordination between this data and the parallel processing code for the MUSCAT on a simulator. The parameter which was employed in the evaluation is shown in Table 2.

In this model, moreover, a secondary cache was not prepared and a case where the main memory has been constituted by a synchronous type DRAM having a standard speed was assumed.

--

Table 2: Simulation Parameters

Item	Parameter
Structure of Various PE's	
Pipe line.	Out-of-order execution from IF, ID, Issue/ Reg to WB.
Instruction Window:	32 instructions for each PE (Integer/ Road store each 16)
Operational Resource	
ALU 2 x PE number L/S Pipe 1 x PE number	
Road/Store:	3 Cycle ... (Translator's Note: Unclear)
Super Scalar Degree:	4 instruction simultaneous decode/complete
Branch Speculation:	Tentative execution up to four branchings
Branch History:	2048 Entry 4 State (Co-ownership between PE's.)

--

Cache

Cache Formula: Instruction/Data separation

Cache Capacity: Each 32 Kbyte (64 byte x 512 entry)

Mapping Method: 4 Way Set Associative LRU Chase-out

5.2. Evaluation Program

An evaluation was carried out by using four kinds of programs including idct, p-block, compress and eqntott. The parallel arrangement of these codes was carried out by using the method of code production shown in Chapter 4, where the successive code was re-written into the parallel code within a range where automatic paralleling was possible by means of a compiler. The paralleling policy will be explained below:

idct:

Two-dimensional reverse discrete cosine transfer treatment. Code extracted from the MPEG2 software decoder. In view of the absence of a dependency relationship between the repetitions, paralelling was carried out by treating one repetition as one thread, and synchroneity obtained at the completion of each dimension. The degree of parallelism became eight as eight repetitions were involved.

p_block:

The existing period of the register between the basic blocks was investigated by using a function (propagate-block) as extracted from SPEC benchmark gcc. There was strong dependency of the data between the basic blocks. Paralleling was carried out by using a method whereby the child thread moves out to the area where there is no problem, with the completion of the parent thread being awaited by a synchronous instruction while the iteration of the for loop was being executed in one thread.

compress:

The file compressed part of 1MB was extracted from SPEC benchmark compress. Paralleling was carried out by using the loop that occupies a majority of the processing as the center. The control flow was dynamically determined and much speculation was employed in view of the existence of much pointer access.

eqntott:

SPEC benchmark eqntott. Most of the processing time was consumed for the processing of the comparative functions of the quick sort. Paralleling was carried out by using speculation.

The characteristics of these codes which have been parallel-treated are shown in Table 3.

(Table 3: Characteristics of Evaluation Program on p. 234.

Program	Overage Number of Instructions	Control Spec.	Data Spec.
idct	140.1	None	None
p-block	99.1	Yes	None
compress	16.3	Yes	Yes
eqtott	11.1	Yes	Yes

In the above, the average number of the instructions means the number of dynamic instruction executions per thread and this does not contain any information about the threads that were cancelled due to the speculation failure.

The dynamic characteristics of the successive execution trace data which were used in the simulation are shown in Table 4.

(Table 4: Characteristics of Trace Data on p. 234.

	Number of Execution Instructions	Branching	Load	Store
idct	32,838	0.2 (%)	15 (%)	10 (%)
p-block	95,941	11	25	12
compress	92,462,899	14	24	9
eqntott	130,723,959	27	22	1

5.3. Evaluation Result

Figure 9 shows the rates of the improvement of the efficiency of each model of 2PE, 4PE and 8PE for IPE model successive code executions.

(Figure 9 on p. 234. Rates of Efficiency Improvement of MUSCAT.)

As parallelism is sufficiently obtained in the case of idct, an improvement of efficiency which is in conformity with the number of PE's is achieved. However, the improvement of efficiency is found to be lower at 8PE. This is believed to be due to the fact that here was a capacity restriction of the data cache and that the number of associations was small as compared with the number of PE's, with the hit ratio coming down by thrushing based on the access competition among the PE's.

In the case of the data caches of 128 Kbytes and 16way construction, both the capacity and the association being four times as large as 1PE, it was possible to realize an improvement of efficiency which was four times for successive executions. As the degree of parallelism was eight, it can be theorized that a further improvement of efficiency can be achieved. Nevertheless, this is restricted by the existence of a successive part on the code and a segregation of the dynamic thread granularity, etc.

Regarding p-block, there is observed an improvement of efficiency up to 4 PE as in the case of idct. In the case of paralleling in p-block, a synchronous instruction PWAIT where the parent thread is awaited at a dependent location or PWAIT has been inserted. Due to the fact that the parallelism has been restricted by the number of threads that had been produced to this point, the result showed that the efficiency is exactly the same at 4PE and 8PE.

In the case of compress and eqntott, the granularity of one thread which is the target of the MUSCAT carried out fine granularity thread paralleling of around 10 instructions. In these codes, the efficiency showed almost no improvement at 2PE but showed an improvement of efficiency around 1.5 times at 4PE. At 2 PE, where the startup and ending were frequent, there was an overhead involving the instructional fetch of a new thread not being able to start until the completion of the final instruction in the thread, thereby inhibiting any improvement in efficiency.

This problem arises even in the case of idct and p-block. In view of the fact that the grain size of the threads is large and the number of thread production/ endings is small, there is hardly any influence upon the efficiency.

In the case of compress, moreover, the rate of efficiency improvement is found small as compared with 4 PE. The reason for this seems to exist in the dependency of the data and a reduction in the rate of the existence of the threads. The rate of thread existence indicates the rate of number of threads that have been completed without being aborted as compared with the threads that have been produced. When this value happens to be high, efficient parallel execution becomes possible. These values will be shown in Table 5.

Table 5: Rate of Existence of Threads (%)

	2PE	4PE	8PE
idct	88.6	87.2	85.9
p-block	86.6	87.0	87.0
compress	43.7	34.3	24.2
eqntott	58.8	58.3	52.7

(Figure 10: Overhead at the Time of Thread startup/ completion. a. Preceding thread. b. Succeeding thread. c. Preceding thread. d. Succeeding thread.)

Table 5 shows the fact that, in the case of compress, the number of the threads that are completed is reduced when the number of PE's increases. As a large number of chldren, grandchildren and great grandchldren are produced in compress, with a result that the number of the abort cases involving the child threads increases, the number of the PE's increases and, at the same time, the existing threads increase in number, with a result that the rate of the existence of the threads comes down. In the case of a problem where the dependency of control is high as in compress, it has become clear that, even when the code producing method shown in Chapter 4 may be employed, the paralleling degree is held around the maximum of four or thereabout.

In the case of eqntott, where the dependency of the control is not as strong as in the case of compress, there is no such tendency, with some improvement in efficiency being confirmed at 8 PE as compared with 4 PE.

In order to compare this MUSCAT control parallel model and the processor model in which the instruction level parallel processing has been strengthened, the rate of the improvement of the efficiency of the super scalar processor intensified model shown in Table 6 was examined. This model has been expanded to the same functional unit number/instruction simultaneous decode number which is equivalent to the case of 2PE, 4PE and 8 PE in the control parallel, including the four kinds of models of 4SS (4 parallel super scalar), 8SS, 16 SS and 32 SS in conformity with the instruction synchronous decoder number. The rate of the efficiency improvement by these models is shown in Figure 11.

--

Table 6: Parameter of Super Scalar Intensified Model.

	4SS	8SS	16SS	32SS
Decode/Cycle	4.	8	16	32
ALU number	2	4	8	16
LS.Pipe Number	1	2	4	8
Instruction Window Size	32	64	128	256
Branching Speculation	4	8	16	32

4SS model has the same parameter as 1 PE in Figure 9.

--

(Figure 11: Rate of the Improvement of Efficiency of the Super Scalar Intensified Model on p. 235.)

The 8SS model which has the same amount of resources as the case of 2PE of the control parallel of compress and eqntott showed an efficiency level which is higher than that of the latter. Otherwise, the MUSCAT model shows higher efficiency than the model where the super scalar processor has been strengthened. The reason for this largely lies in the fact that the code producing method which was shown in Chapter 4 made it possible for those whose execution is certain or whose execution probability is high to be started up as the threads on a priority basis.

Regarding the factors that obstruct the improvement of the efficiency of the strengthened model of the super scalar, the instruction window size is limited to the range of one repetition of the loop. It is believed that this problem could be solved by effectively carrying out the loop unrolling; however, the problem of the number of logic registers that can be designated by an instruction restricting the number of unrollings remains.

In the case of compress and eqntott where the dependency of the control between the basic blocks is strong, another reason lies in a high cost of the re-execution that accompanies a failure of the branch speculation.

In order to confirm the effect of the branching failures, the 32 SS model and the 4SS model of the compress were executed with the rate of the success of branching anticipations at 100 per cent. As a result, an improvement of efficiency which was 1.61 times that of 4SS model was confirmed in the 32SS model.

6. Conclusion

As a result of the recent evaluation, the following points have been clarified:

. In the case of compress and eqntott, it is possible to realize high speed even where speed improvement is difficult to achieve by the other paralleling methods due to the strong dependency of the control. The reason for this lies in the advanced execution of the threads where execution is carried out with certainty or at a high accuracy rate. In the case of the overhead, where division is effected into the threads of fine granularity, the problem can be solved by the high speed thread startup and the fork-time succession of the register.

. The problems of idct and p.block where the granularity is large can also be efficiently solved. Even though the dependency among the instructions exists in the range of the instruction windows, it can effectively utilize the character that, if the range is expanded, it becomes possible to execute the instructions without the existence of dependency among the instructions.

. In the case of the fine granularity thread processing which is targeted by control parallelism, the serious problem based on the fork one-time model was not observed.

In the super scalar system, on the other hand, the efficiency is limited by the anticipated accuracy of branching in the case where the dependency of control is strong. Either the branching anticipation of a higher degree or the introduction of a conditional execution instruction is conceivable for the purpose of improving the parallelism per one cycle. However, it is impossible to improve the rate of the successful branching anticipations to 100 per cent and the branchings where the conditional execution instruction can be used are also limited.

In the case of those problems where the granularity is large, moreover, the code scheduling that stores those instructions which can be simultaneously issued in the instruction window becomes important. Nevertheless, a large-scale movement of the instructions accompanies such problems as the analytical ability of the compiler, the number of the logic registers, and the insertion of the compensation code that accompanies the large-area movement.

From what has been described above, it has become clear that the control parallel processing and the MUSCAT are capable of becoming a powerful choice as the next generation microprocessor architecture for instruction level parallel processing.

On the other hand, it has become clear that there are such problems as will be described below:

. In the case where a fine granular thread is executed with a small-number PE model, the improvement of efficiency is inhibited as the startup of the next thread cannot be achieved unless the thread is completely completed.

. When the number of PE's has increased, thrushing takes place due to a shortage of the capacity of the cache or the deficiency of the associations in number, with a result that the improvement of the efficiency is inhibited.

Regarding the former problem, it can be solved by starting the decoding of the succeeding thread subsequent to the final instruction decoding of the preceeding thread. However, thread management becomes complicated in this case.

7. Related Researches:

The Multiscalar (3) of the University of Wisconsin can be mentioned as a representative of the control parallel processing of such fine granular threads. The Multiscalar is capable of dynamically selecting the thread to be forked on a selective basis, thereby making it possible to effect the parallel execution, at high efficiency, of such problems on which action is difficultly anticipated on a stationary basis. Nevertheless, there are such problems as the complicated thread management or the increase in the information that accompanies the code in connection with paralleling.

As the register succession after the fork becomes possible, moreover, it becomes possible to raise the freedom of the threads; however, this brings about a further complication in register management. In view of the need to designate the succession information in detail about the register to the code, there will be an increase in the length of the wording.

As the data dependency on the memory is all handled speculatively, a speculative execution which does not contribute toward the improvement of the efficiency is carried out in the case where there clearly is the dependency of the address.

In the case of SPSM architecture (4) which has been proposed by IBM, a thread production instruction and a thread awaiting instruction are incorporated into a single program and a future thread that anticipates the execution by means of a thread production instruction is produced, thereby carrying out the anticipated execution.

The future thread executes up to the point of a thread awaiting instruction and wait for the arrival of the original execution, thereby merging the results. The multiple fork of one thread becomes possible and the freedom of the code production becomes higher. In view of the complication of the thread management and as the register value will finally merge with the real execution thread in the future thread, there will be a problem in which waiting is required until a register that will not be referred to in the future is confirmed.

8. Conclusion

Currently, a basic evaluation of the MUSCAT has just been completed. In the future, we will continue the examination of the compiler and a detailed evaluation thereof through an architectural improvement with an eye toward a further improvement of the efficiency on the basis of what has been clarified this time. It is also our intention to examine the exceptional cases or hardware materialization.

References:

(1) Motomura et al.: "Proposal of an Ordered Multi-thread Execution Model," JSPP 95, pp. 99-106, May 1995.

(2) Torii et al.: "A Programming Model of Ordered Multi-thread Architecture and Its Evaluation," JSPP 96, pp. 299-306, June 1996

(3) Gurinder S. Sohi, Scott E. Breach and T. N. Vijaykumar, "Multiscalar Processor," The 22nd International Symposium on Computer Architecture, IEEE Computer Society Press, 1995, pp. 414-425

(4) Pradeep K. Dubey, Kevin O'Brien, Kathryn M. O'Brien, Charles Barton: "Single-Program Speculative Multithreading (SPSM) Architecture: Compiler-Assisted Fine-Grained Multi-threading," Parallel Architectures and Compilation Techniques, IFIP 1995, pp. 109-121.